

Parallelizing the Chambolle Algorithm for Performance-Optimized Mapping on FPGA Devices

IVAN BERETTA, École Polytechnique Fédérale de Lausanne

VINCENZO RANA, Politecnico di Milano

ABDULKADIR AKIN, École Polytechnique Fédérale de Lausanne

ALESSANDRO ANTONIO NACCI and DONATELLA SCIUTO, Politecnico di Milano

DAVID ATIENZA, École Polytechnique Fédérale de Lausanne

The performance and the efficiency of recent computing platforms have been deeply influenced by the widespread adoption of hardware accelerators, such as graphics processing units (GPUs) or field-programmable gate arrays (FPGAs), which are often employed to support the tasks of general-purpose processors (GPPs). One of the main advantages of these accelerators over their sequential counterparts (GPPs) is their ability to perform massive parallel computation. However, to exploit this competitive edge, it is necessary to extract the parallelism from the target algorithm to be executed, which generally is a very challenging task.

This concept is demonstrated, for instance, by the poor performance achieved on relevant multimedia algorithms, such as Chambolle, which is a well-known algorithm employed for the optical flow estimation. The implementations of this algorithm that can be found in the state of the art are generally based on GPUs but barely improve the performance that can be obtained with a powerful GPP. In this article, we propose a novel approach to extract the parallelism from computation-intensive multimedia algorithms, which includes an analysis of their dependency schema and an assessment of their data reuse. We then perform a thorough analysis of the Chambolle algorithm, providing a formal proof of its inner data dependencies and locality properties. Then, we exploit the considerations drawn from this analysis by proposing an architectural template that takes advantage of the fine-grained parallelism of FPGA devices. Moreover, since the proposed template can be instantiated with different parameters, we also propose a design metric, the expansion rate, to help the designer in the estimation of the efficiency and performance of the different instances, making it possible to select the right one before the implementation phase. We finally show, by means of experimental results, how the proposed analysis and parallelization approach leads to the design of efficient and high-performance FPGA-based implementations that are orders of magnitude faster than the state-of-the-art ones.

Categories and Subject Descriptors: B.6.1 [Design Styles]: Parallel Circuits; I.4.8 [Scene Analysis]: Motion; C.1.3 [Other Architecture Styles]: Data-Flow Architectures; B.8.2 [Performance Analysis and Design Aids]

General Terms: Design, Algorithms, Performance

This work was partially supported by ONR-G grant N62909-14-1-N072 and the E4Bio RTD project (200021_159853) evaluated and financed by the Swiss NSF.

Authors' addresses: I. Beretta and D. Atienza, EPFL STI IEL ESL, ELG 131 (Bâtiment ELG), Station 11, CH-1015 Lausanne, Switzerland; emails: {ivan.beretta, david.atienza}@epfl.ch; V. Rana, A. A. Nacci, and D. Sciuto, Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Piazza Leonardo da Vinci 32, 20133 Milano; emails: {vincenzo.rana, alessandro.nacci, donatella.sciuto}@polimi.it; A. Akin, EPFL STI IEL LSM, ELD 333 (Bâtiment ELD), Station 11, CH-1015 Lausanne, Switzerland; email: abdulcadir.akin@epfl.ch.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2016 ACM 1539-9087/2016/03-ART44 \$15.00

DOI: <http://dx.doi.org/10.1145/2851497>

Additional Key Words and Phrases: Chambolle, optical flow, $TV-L^1$, field programmable gate arrays, parallel architectures, custom hardware

ACM Reference Format:

Ivan Beretta, Vincenzo Rana, Abdulkadir Akin, Alessandro Antonio Nacci, Donatella Sciuto, and David Atienza. 2016. Parallelizing the Chambolle algorithm for performance-optimized mapping on FPGA devices. *ACM Trans. Embed. Comput. Syst.* 15, 3, Article 44 (March 2016), 27 pages.
DOI: <http://dx.doi.org/10.1145/2851497>

1. INTRODUCTION

Heterogeneous and specialized computation is forecast to increasingly grow over the coming years and establish itself as one of the main paradigms for embedded systems design [Cordes et al. 2013]. The employment of special-purpose cores to perform a complex functionality within a system-on-chip (SoC) is motivated by higher performance and lower power consumption with respect to an equivalent execution on a general-purpose processing unit. Furthermore, in certain domains, such as multimedia processing, these specialized cores perform tasks that are sufficiently general to guarantee a good reusability in a wide range of systems. For example, specialized cores can be used to accelerate common operations such as convolution filters [Jamro and Wiatr 2001] or the Jacobi operator [Sleijpen and Vorst 2000].

The design of special-purpose hardware modules traditionally aims at optimizing their computational efficiency while meeting predefined area requirements that may be imposed when the core is part of a more complex multicore SoC. To achieve the target performance, application-specific accelerators can be implemented on different cutting-edge platforms, such as graphics processing units (GPUs) or field-programmable gate arrays (FPGAs). However, even though GPUs are faster than FPGAs, they show a rigid structure designed for *single instruction multiple data* processing, and hence they are not a good choice when dealing with algorithms with very complex data dependencies among iterations [Bodily et al. 2010]. FPGAs, on the other hand, provide a fully customizable platform where any kind of custom operation, either complex or very simple, can be implemented in hardware and applied on multiple blocks of data in parallel. Unfortunately, the design of complex and custom FPGA systems is a very challenging task, and tools to drive the designer in the definition of such architectures are still not mature.

Representative examples of important computation-intensive algorithms that greatly benefit from parallelization and performance optimization can be found in the field of multimedia processing [Jian et al. 2013; Ali et al. 2014]. Several researchers have addressed their effort toward some of these algorithms in the past years [Chen et al. 2012; Ghodhbani et al. 2014]. In this article, we focus our attention to Chambolle [2004], which is a relevant algorithm belonging to this class and for which a high-performance parallel implementation has not yet been proposed, as we show in the analysis of the state-of-the-art approaches presented in Section 2.

The Chambolle algorithm is a well-known and widely employed algorithm in such fields as motion estimation and compensation, or rolling shutter correction (see Section 2 for more details). However, even though this algorithm is used in many applications (e.g., the $TV-L^1$ optical flow estimation described in Section 2), no parallel and efficient implementation has been proposed so far; in fact, even the best-performing implementations on GPUs are essentially sequential, and they do not achieve real-time frame rates with high-resolution images [Zach et al. 2007]. This lack of performance is mainly due to the complex data dependencies schemas that usually characterize this kind of algorithm. In addition to the lack of efficient GPU and multicore implementations, no hardware implementation methodology exists to exploit the high amount of resources available on the latest programmable devices, such as FPGAs. For these reasons, we believe that the Chambolle algorithm can be considered as a cornerstone

for many multimedia systems that deal with challenging problems (e.g., the optical flow estimation [Behbahani et al. 2007]) and for which efficient implementations have not yet been found, mainly because of their complex data dependencies.

This work builds upon the Chambolle implementation that we first outlined in Akin et al. [2011], complementing it with a more detailed algorithm analysis, as well as a deep design space exploration. Specifically, we propose a breakdown of the Chambolle kernel, formally proving its dependency pattern and its locality. We then define a novel algorithmic-level metric to drive the design space exploration of iterative algorithms, which we named *expansion rate*. The metric enables one to estimate implementation aspects, such as the impact of memory transfers, as a function of the geometry of the algorithm. Finally, we extend the design space exploration to other platforms, specifically to GPUs.

The remainder of this article is structured as follows. In Section 3, we provide a detailed analysis of the Chambolle algorithm, focusing on its main characteristics and proprieties. In Section 4, we describe the proposed design strategy to efficiently tackle its complexity, parallelizing its computation to drastically improve its performance. After showing the proposed architectural template, we introduce the concept of expansion rate, another relevant contribution of this work. Section 5 reports the design space exploration for the Chambolle algorithm and presents the implementation aspects of the proposed hardware implementation. Section 6 describes the experimental results proving that the proposed parallelization of the Chambolle algorithm is considerably faster than the solutions found in the literature. These approaches are mainly based on GPU acceleration that do not completely exploit the implicit fine-grained parallelism of this kind of multimedia algorithm. Section 7 shows how the proposed approach, based on a finer parallelization of the input algorithm and targeting FPGA devices, is able to drastically increase the degree of parallelism that can be extracted from the algorithm, exploiting it to increase the efficiency and the performance of the computing architecture. Section 8 concludes the article by drawing some final considerations.

2. STATE OF THE ART

The *optical flow* is a vector field representing the movement of an object in a sequence of frames, and it can be determined by analyzing the variation of the brightness inside a sequence of successive images [Verri and Poggio 1989]. The estimation of this vector field is one of the most important problems in image and video processing, as it can be employed for motion estimation [Sun et al. 2000] and compensation [Lin et al. 1997], as well as in other fields, such as robotics [Kim et al. 2007] and even medical analysis [Behbahani et al. 2007]. Another important application of the optical flow is the correction of an image acquired by CMOS optical sensors using the rolling shutter technique [Baker et al. 2010], which is used today in most of the low-end photo cameras. In particular, rolling shutter is a method of image acquisition in which each frame is recorded by scanning across the frame either vertically or horizontally, which may generate errors and distortions in the final image.

The optical flow estimation is a computationally challenging problem [Behbahani et al. 2007] because of the large amount of movements that can be detected in a frame, and because of the noise that can alter the image brightness. A wide range of different techniques, such as those of Horn and Schunck [1981], Black and Anandan [1993], and Papenberger et al. [2006], have been proposed in the past, but *variational* methods [Aubert et al. 1999] (i.e., algorithms based on the minimization of a quantity known as total variation [Rudin et al. 1992]) have emerged as one of the most successful approaches in recent years. The variational technique that we consider in this work is called $TV-L^1$ [Pock et al. 2007], which distinguishes itself from other approaches because it can handle highly varying intensities in the frames.

The TV- L^1 method includes both a mathematical definition of the variational problem and a numerical scheme to compute the solution. The numerical scheme is based on a fixed-point algorithm originally proposed by Chambolle [2004], which iteratively refines the solution (which in this case represents the optical flow estimation) at different levels of precision. Although TV- L^1 seems to be very promising from a theoretical point of view, its implementations fail to reach real-time performance (i.e., to process at least 30 frames per second (fps)), except for very small images. A multithread software implementation of TV- L^1 that has been developed and analyzed at EPFL, for example, can take more than 15 seconds to process just one frame on a standard x86 workstation, and up to 50 seconds are required on the ARM processor of an Apple iPhone 3GS. The profiling of the estimations of the TV- L^1 optical flow on both platforms shows that the Chambolle algorithm itself is the bottleneck that generates the poor timing performance. In fact, besides the execution of an outermost loop that does not require any complex matrix operation, approximately 90% of the execution time is spent on the Chambolle iterative technique, which proves to be the most critical and computationally intensive part.

However, all implementations of the Chambolle algorithm that can be found in the literature fail in achieving real-time frame rates with high-resolution images [Zach et al. 2007]. Furthermore, to the best of our knowledge, a parallel implementation of this approach has never been proposed because of the complex dependencies among the intermediate results [Akin et al. 2011].

In Pock et al. [2007] and Zach et al. [2007], the robust TV- L^1 technique to calculate the optical flow between two frames is proposed and implemented using modern GPUs. The authors proved that a real-time frame rate can be achieved by the most powerful devices for low-resolution sequences, but only very few frames that are larger than 512×512 can be processed in 1 second. A Matlab implementation of the technique in Zach et al. [2007] requires 5 to 6 seconds to complete the estimation of the optical flow on a high-end workstation, and it also shows some limitations in terms of memory usage.

Additional hardware results of the estimation of the TV- L^1 optical flow on GPUs can be also found in Weishaupt et al. [2010], but even the fastest implementation cannot top a rate of 6fps, even on 512×512 images. A full summary of the performance of the aforementioned state-of-the-art implementations of Chambolle are reported in Section 6 as a reference to evaluate the solutions proposed in this article.

Fast estimations of the optical flow can be achieved by using different techniques and by simplifying the working domain. For example, the implementation proposed in Abutaleb et al. [2009] can process up to 156fps on 768×576 images, working on a low-cost FPGA device. However, the resulting optical flow is specifically suited for motion detection, and it cannot be used in other applications, such as rolling shutter correction. The specific target allows the authors to filter the input frames and in particular to apply background subtraction, which heavily simplifies the amount of data to be processed for the optical flow estimation.

3. CHAMBOLLE ALGORITHM ANALYSIS

This section presents the analysis that we performed on the Chambolle algorithm, describing the structure of its *dependency schema* (Section 3.2) and providing a formal proof of its *locality* (Section 3.3). The notation used in this section is a minor modification of the one used in Chambolle [2004] and requires a few basic concepts that are described in Section 3.1. Finally, Section 3.4 presents a simplified pseudocode formulation of the Chambolle algorithm.

3.1. Preliminary Definitions

In the context of multimedia processing, the input of the Chambolle algorithm is represented as a rectangular matrix of length L and width W , which represents a picture

of $L \times W$ pixels. Let X be defined as the Euclidean space $X = \mathbb{R}^{L \times W}$, and let Y be the Cartesian product $Y = X \times X$. Finally, let us recall the definition of the *Euclidean norm* $\| \cdot \|$ over \mathbb{R}^2 , which is defined as $\| y \| = \sqrt{y_1^2 + y_2^2}$, for any point $y = (y_1, y_2) \in \mathbb{R}^2$.

It is now possible to introduce the two main operators that are used in the formulation of the Chambolle algorithm: the discrete gradient divergence operators. Given an element $x \in X$, the *discrete gradient* $\nabla x \in Y$ is defined as

$$(\nabla x)_{i,j} = \left((\nabla x)_{i,j}^{(1)}, (\nabla x)_{i,j}^{(2)} \right), \quad (1)$$

where

$$(\nabla x)_{i,j}^{(1)} = \begin{cases} x_{i+1,j} - x_{i,j}, & \text{if } i < L \\ 0, & \text{if } i = L \end{cases}, \quad (\nabla x)_{i,j}^{(2)} = \begin{cases} x_{i,j+1} - x_{i,j}, & \text{if } j < W \\ 0, & \text{if } j = W \end{cases} \quad (2)$$

for $i = 1, \dots, L$ and $j = 1, \dots, W$. The cases $i = L$ and $j = W$ are considered separately, as they refer to pixels that lie on the boundaries of the matrix.

The *discrete divergence* operator takes an element $p \in Y$ as an operand and returns the value $\text{div } p \in X$, defined as

$$(\text{div } p)_{i,j} = \begin{cases} p_{i,j}^{(1)} - p_{i-1,j}^{(1)}, & \text{if } 1 < i < L \\ p_{i,j}^{(1)}, & \text{if } i = 1 \\ -p_{i-1,j}^{(1)}, & \text{if } i = L \end{cases} + \begin{cases} p_{i,j}^{(2)} - p_{i,j-1}^{(2)}, & \text{if } 1 < j < W \\ p_{i,j}^{(2)}, & \text{if } j = 1 \\ -p_{i,j-1}^{(2)}, & \text{if } j = L. \end{cases} \quad (3)$$

As discussed in the previous sections, the Chambolle algorithm aims at minimizing a quantity known as *total variation* [Rudin et al. 1992]. With the concepts defined in this section, it is now possible to formalize this metric. Given $g \in X$ and $\theta > 0$, the minimization of the total variation can be formulated as follows:

$$\min_{x \in X} \left\{ \frac{\| x - g \|^2}{2\theta} + \sum_{1 \leq i \leq L, 1 \leq j \leq W} \| (\nabla x)_{i,j} \| \right\}. \quad (4)$$

As shown in Chambolle [2004], the minimization problem has a closed-form solution whose analytical equation is known, but its numerical estimation is not straightforward. To find a solution numerically, the problem must be expressed in the following form:

$$\min_{p \in Y} \{ \| \theta \text{div } p - g \|^2 : \| p_{i,j} \|^2 \leq 1, \quad \forall i = 1, \dots, L, j = 1, \dots, W \}. \quad (5)$$

This formulation can be numerically approached using a recursive technique known as *semi-implicit gradient descent* [Chambolle 2004], which is the core part of the Chambolle algorithm. In particular, for any $n \geq 0$, which defines number of iterations or *levels*, an element $p \in Y$ is recursively adjusted as follows:

$$p_{i,j}^{(n+1)} = \frac{p_{i,j}^{(n)} + \tau (\nabla \Phi^{(n)})_{i,j}}{1 + \tau \| (\nabla \Phi^{(n)})_{i,j} \|}, \quad \Phi^{(n)} = \text{div } p^{(n)} - \frac{g}{\theta}, \quad (6)$$

where $\tau > 0$ is a fixed value (in general, it is equal to 1/4 to guarantee the convergence of the algorithm [Chambolle 2004]), and $p^{(0)} = 0$ by definition. The matrix $\Phi^{(n)} \in X$ is a matrix that is defined to keep the notation compact.

3.2. Dependency Schema

According to Equation (6), the solution of the Chambolle algorithm recursively depends on previous values (e.g., there is an explicit dependency between $p_{i,j}^{(n+1)}$ and $p_{i,j}^{(n)}$), which

may prevent a parallelized implementation because a large amount of data might be required to compute the value of $p_{i,j}^{(n+1)}$. The goal of this section is to unroll the dependencies included in Equation (6) and derive the full shape of the stencil.

For the sake of illustration, the points on the boundaries of the matrices are omitted, and therefore indices i and j are always strictly greater than 1 and strictly lower than L and W , respectively. In fact, boundary values are only a special case of the proposed analysis, and they can be easily handled by substituting the corresponding values from Equations (2) and (3).

In Equation (6), the denominator is a scalar quantity, whereas both of the two terms in the numerator belong to $Y = X \times X$. As a consequence, $p_{i,j}^{(n+1)} \in Y$, and thus it can be written as

$$p_{i,j}^{(n+1)} = \left(px_{i,j}^{(n+1)}, py_{i,j}^{(n+1)} \right), \quad (7)$$

where both $px^{(n+1)}$ and $py^{(n+1)}$ are $L \times W$ matrices computed at level $n + 1$.

The term $(\nabla \Phi^{(n)})_{i,j}$ can then be unrolled according to Equations (1) and (2), remembering that the point (i, j) is not on the boundaries of the matrix, and obtaining

$$(\nabla \Phi^{(n)})_{i,j} = \left((\nabla \Phi^{(n)})_{i,j}^1, (\nabla \Phi^{(n)})_{i,j}^2 \right) = \left(\Phi_{i+1,j}^{(n)} - \Phi_{i,j}^{(n)}, \Phi_{i,j+1}^{(n)} - \Phi_{i,j}^{(n)} \right). \quad (8)$$

By substituting this result in Equation (6), and by considering the decomposition of $p_{i,j}^{(n+1)}$ shown in (7), two separate equations for $px_{i,j}^{(n+1)}$ and $py_{i,j}^{(n+1)}$ can be written:

$$px_{i,j}^{(n+1)} = \frac{px_{i,j}^{(n)} + \tau(\Phi_{i+1,j}^{(n)} - \Phi_{i,j}^{(n)})}{1 + \tau \parallel (\nabla \Phi^{(n)})_{i,j} \parallel}, \quad (9)$$

$$py_{i,j}^{(n+1)} = \frac{py_{i,j}^{(n)} + \tau(\Phi_{i,j+1}^{(n)} - \Phi_{i,j}^{(n)})}{1 + \tau \parallel (\nabla \Phi^{(n)})_{i,j} \parallel}. \quad (10)$$

Finally, $\Phi^{(n)}$ should be expressed as a function of $px^{(n)}$ and $py^{(n)}$. This can be achieved by computing the $\text{div } p^{(n)}$ term according to Equation (3),

$$(\text{div } p^{(n)})_{i,j} = px_{i,j}^{(n)} - px_{i-1,j}^{(n)} + py_{i,j}^{(n)} - py_{i,j-1}^{(n)}, \quad (11)$$

and thus getting that an element $\Phi_{i,j}^{(n)}$ can be expressed as

$$\Phi_{i,j}^{(n)} = \left(\text{div } p^{(n)} - \frac{g}{\theta} \right)_{i,j} = px_{i,j}^{(n)} - px_{i-1,j}^{(n)} + py_{i,j}^{(n)} - py_{i,j-1}^{(n)} - \frac{g_{i,j}}{\theta}. \quad (12)$$

The resulting value is substituted into Equations (9) and (10) to show the dependency between $px^{(n+1)}$ and $py^{(n+1)}$ and some points in $px^{(n)}$ and $py^{(n)}$ (i.e., points referring to the previous iteration). In particular, the resulting equations are

$$\begin{aligned} px_{i,j}^{(n+1)} = & \frac{px_{i,j}^{(n)} + \tau[px_{i+1,j}^{(n)} - 2px_{i,j}^{(n)} + px_{i-1,j}^{(n)}]}{1 + \tau \parallel (\nabla \Phi^{(n)})_{i,j} \parallel} \\ & + \frac{\tau[py_{i+1,j}^{(n)} - py_{i,j}^{(n)} + py_{i,j-1}^{(n)} - py_{i+1,j-1}^{(n)} + \left(\frac{g_{i,j} - g_{i+1,j}}{\theta} \right)]}{1 + \tau \parallel (\nabla \Phi^{(n)})_{i,j} \parallel}, \end{aligned} \quad (13)$$

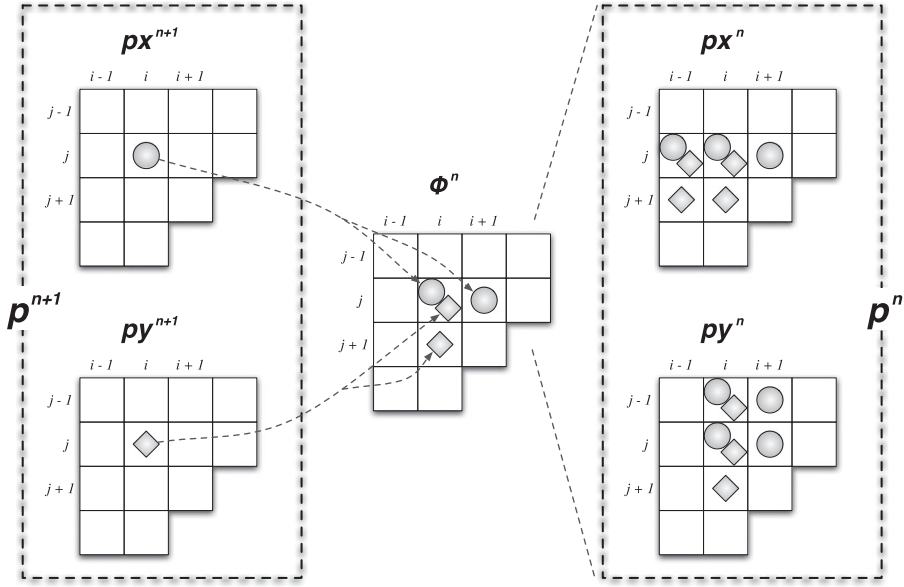
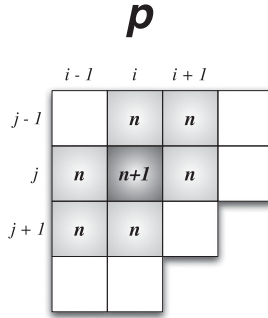
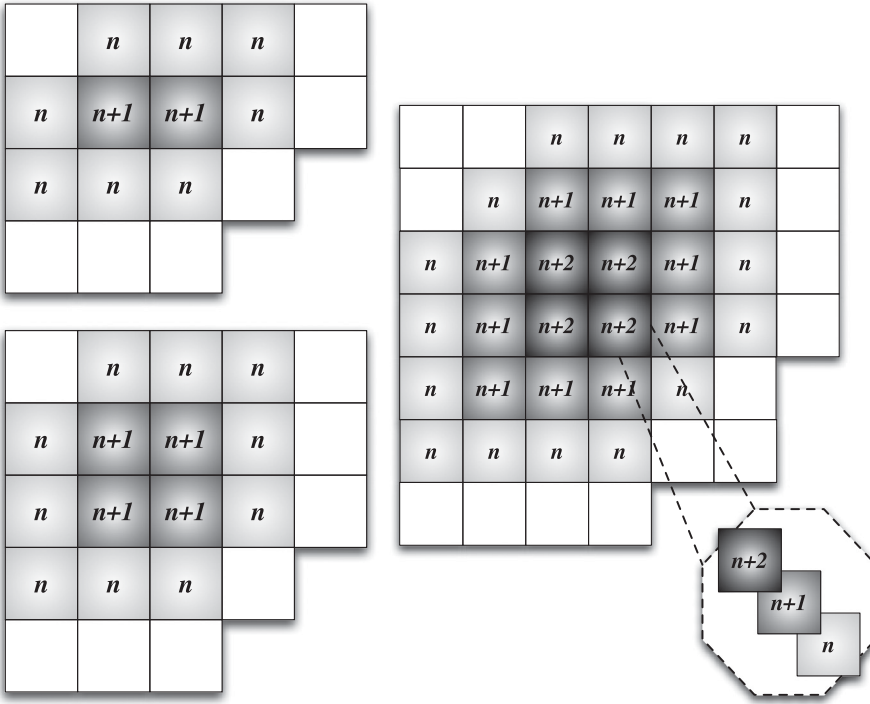
(a) Dependencies among matrices $px^{(n+1)}$, $py^{(n+1)}$, $\Phi^{(n)}$, $px^{(n)}$, and $py^{(n)}$ (b) Simplified representation of the dependencies among $p^{(n+1)}$ and $p^{(n)}$

Fig. 1. Graphical representation of the stencil shape of the Chambolle algorithm.

$$\begin{aligned}
 py_{i,j}^{(n+1)} = & \frac{py_{i,j}^{(n)} + \tau [px_{i,j+1}^{(n)} - px_{i,j}^{(n)} + px_{i-1,j}^{(n)} - px_{i-1,j+1}^{(n)}]}{1 + \tau \| (\nabla \Phi^{(n)})_{i,j} \|} \\
 & + \frac{\tau [py_{i,j+1}^{(n)} - 2py_{i,j}^{(n)} + py_{i,j-1}^{(n)} + (\frac{g_{i,j} - g_{i,j-1}}{\theta})]}{1 + \tau \| (\nabla \Phi^{(n)})_{i,j} \|}.
 \end{aligned} \tag{14}$$

A visual representation of the dependencies extracted from Equations (13) and (14) is shown in Figure 1(a), where all of the intermediate matrices $px^{(n+1)}$, $py^{(n+1)}$, $px^{(n)}$, $py^{(n)}$, and $\Phi^{(n)}$ are illustrated. However, since $px^{(n)}$ and $py^{(n)}$ are only known if the element $p = (px, py)$ is known, it is possible to use a more compact representation that only considers $p^{(n+1)}$ and $p^{(n)}$, thus obtaining the schema in Figure 1(b). Since Figure 1(b) depicts the



(a) Dependencies for the computation of two and four points of $p^{(n)}$

(b) Dependencies for the computation of multiple levels (n to $n+2$)

Fig. 2. Generalization of the dependencies among the points in matrix p .

dependencies between two consecutive iterations, it also graphically illustrates the shape of the stencil applied by the Chambolle algorithm.

3.3. Locality of the Algorithm

The stencil shown in Figure 1(b) can be generalized in two ways. First, it is possible to identify the dependencies when more than one element of the matrix has to be computed, such as a submatrix of $p^{(n+1)}$ of size $l \times w$. Figure 2(a) shows the dependency schema when 2×1 and 2×2 submatrices are computed at level $n+1$. Second, it is possible to increase the number of levels beyond $n+1$, as shown in Figure 2(b) for level $n+2$.

In general, a submatrix of size $l \times w$ at level $n+1$ depends on the same $l \times w$ pixels at level n , but it also requires a ring of additional elements at level n that surrounds the submatrix. In the example with a 2×2 submatrix shown in Figure 2(a), the goal is to compute four points at level $n+1$, which can be achieved starting from the same points at level n , and including a ring of 10 elements at level n that surrounds the submatrix (notice that the pixels in the upper-left and in the lower-right corners are not required). Similarly, if more levels are considered at once, the elements of the ring require additional surrounding points, thus leading to a dependency schema composed of concentric rings of growing size, as shown in Figure 2(b).

Given the regularity of the dependency schema, it is possible to estimate the number of points that are required to compute a generic submatrix at an arbitrary level. Let $\Omega(l, w, N)$ be the number of elements needed to calculate a submatrix of size $l \times w$ (with

$1 \leq l \leq L$ and $1 \leq w \leq W$) at a level $N \geq 2$. It can be observed that the case $N = 1$ is trivial, as no recursion is necessary to get the result. In addition, if a point at level N has to be computed, all values from level $N - 1$ to level 1 must be known so that the recursion of Equation (6) will terminate. In the case of Chambolle, the value of $\Omega(l, w, N)$ can be computed as follows:

$$\Omega(l, w, N) = \sum_{k=1}^{N-1} \left[(l + 2k)(w + 2k) - 2 \sum_{h=1}^k h \right]. \quad (15)$$

The outermost summation considers all levels $N - k$ and computes the number of points that are required at that level. At each level, both the length and the width of the surrounding ring enlarge by two points, an effect that is captured by the $(l + 2k)(w + 2k)$ term. The innermost summation corrects the estimation by removing a level-dependent number of points from the upper-left and the lower-right corners of the ring, which are not required at that level. For example, let us consider the computation of a 2×2 submatrix at level $N = 3$, which is the same schema shown in Figure 2(b) when $n = 1$. For $k = 1$, level $N - k = 2$ is considered, and the number of points that are required is equal to $(2 + 2 \cdot 1)(2 + 2 \cdot 1) - 2 \cdot 1 = 14$. At $k = 2$, level $N - k = 1$ is considered, and a total of $(2 + 2 \cdot 2)(2 + 2 \cdot 2) - 2 \cdot 3 = 30$ points are needed. Overall, $14 + 30 = 44$ points are required to compute a 2×2 submatrix at level 3.

The value of $\Omega(l, w, N)$ can be used to compute the SER metric of Chambolle that will be introduced in Section 4. It is also important to remark that in general, $\Omega(l, w, N)$ can be considered as an upper bound of the total number of pixels, because some of the points may be located on the boundaries of the matrix, so they depend on a smaller number of neighbors. Conversely, $\Omega(l, w, N)$ is an exact estimation when the points are not located on the matrix borders. In both cases, the fact that the number of required neighbors is bounded by $\Omega(l, w, N)$ ensures that this computation can be performed locally.

3.4. A Simplified Pseudocode Formulation of Chambolle

In the previous sections, the locality of the Chambolle algorithm and its dependency schema has been analyzed starting from its mathematical formulation. For the sake of clarity, a simpler pseudocode formulation of the algorithm is now introduced. The pseudocode form was first proposed in Zach et al. [2007], and it introduces a set of high-level macrooperations that are better suited for hardware design while preserving the same dependencies underlined in Figure 2.

In the pseudocode formulation, the optical flow between the two input frames I_0 and I_1 —both expressed in a matrix form—is represented by a bi-dimensional vector $u = (u_1, u_2)$, which is the output of the Chambolle algorithm. The vector u is initialized at 0, and its final value is computed by means of an iterative sequence of levels, as discussed in the previous sections. At each level, a support variable $v = (v_1, v_2)$ is defined using a thresholding function of I_1 and of the value of u computed at the previous level [Zach et al. 2007]. Then, the value of u at the current level is determined using the iterative steps of the Chambolle algorithm, which are reported in Algorithm 1. For the sake of simplicity, the pseudocode only shows the computation of u_1 , but u_2 is computed in the same way, by simply substituting u_1 and v_1 with u_2 and v_2 .

The vector u is updated by means of two intermediate values, namely $px = (px_{u1}, px_{u2})$ and $py = (py_{u1}, py_{u2})$, which are initialized at 0 [Zach et al. 2007]. To simplify the description, the auxiliary variables $Term$, $Term_1$, and $Term_2$ are also introduced to store the intermediate results of the computation (lines 3 through 5). The $Backward_X(z)$ function returns a matrix where each element of z is subtracted by its left neighbor, whereas in $Backward_Y$ it is subtracted by its upper neighbor. Similarly, in

ALGORITHM 1: Chambolle Algorithm

```

1: for  $i = 1, \dots, N_{iterations}$  do
2:    $\text{div } p = (\text{Backward}_X(px_{u1}) + \text{Backward}_Y(py_{u1}))$ 
3:    $\text{Term} = \text{div } p - v_1/\theta$ 
4:    $\text{Term}_1 = \text{Forward}_X(\text{Term})$ 
5:    $\text{Term}_2 = \text{Forward}_Y(\text{Term})$ 
6:    $|\nabla u_1| = \sqrt{\text{Term}_1^2 + \text{Term}_2^2}$ 
7:    $px_{u1} = [px_{u1} + \tau/\theta \cdot \text{Term}_1]/[1 + \tau/\theta \cdot |\nabla u_1|]$ 
8:    $py_{u1} = [py_{u1} + \tau/\theta \cdot \text{Term}_2]/[1 + \tau/\theta \cdot |\nabla u_1|]$ 
9:    $u_1 = v_1 - \theta \cdot \text{div } p$ 
10: end for

```

function Forward_X the element is subtracted by its right neighbor and in Forward_Y by its lower neighbor. It is worth noting that according to the way they are invoked in Algorithm 1, these four functions generate the same stencil shape illustrated in Figure 2. Finally, the constants θ and τ are the same values that are used in the mathematical formulation of Chambolle and determine the precision of the algorithm.

4. THE PROPOSED DESIGN STRATEGY

The analysis described in Section 3 shows that the Chambolle algorithm is characterized by the following properties:

- (1) No *read-after-write* (RAW) conflicts exist within a single iteration, as shown by the pseudocode presented in Algorithm 1. This means that the computation of an element at iteration $i + 1$ cannot depend on the value of another element at iteration $i + 1$, but only on previously generated elements, such as those computed at iteration i .
- (2) As shown in Section 3.3, which describes the *locality* of the Chambolle algorithm, the set of elements required to compute an element at the iteration $i + 1$ is a small subset of the frame f_i produced at the i -th iteration, and these elements are spatially close to element p that has to be computed.
- (3) Finally, the analysis of the dependency schema of the Chambolle algorithm performed in Section 3.2 shows that given two target elements separated by a translation, the corresponding dependency schemas have the same shape, but they are translated by the same distance as the target element.

By exploiting these features, we have been able to propose an efficient architecture that serves as a template for the high-performance and parallel implementation of the Chambolle algorithm, as described in Section 4.1. Since the template has to be tailored to the specific needs of the designer (i.e., to explore the resource-performance trade-offs), in Section 4.2 we introduce a set of metrics that can be used by the designer to tune the different architectural parameters of the proposed template.

4.1. Proposed Architectural Template

The proposed architectural template is based on a computational structure that is different from the straightforward one-entire-frame-at-a-time approach. In fact, it aims at directly computing a portion of the results of an arbitrary iteration, by loading and processing only the elements that are required to produce the output, according to the dependencies schema of the algorithm. The set of elements produced as an output are typically a subset of the elements that are processed as an input because of data dependencies, and therefore the core that performs such multi-iteration computation can be seen as a *cone* (Figure 3).

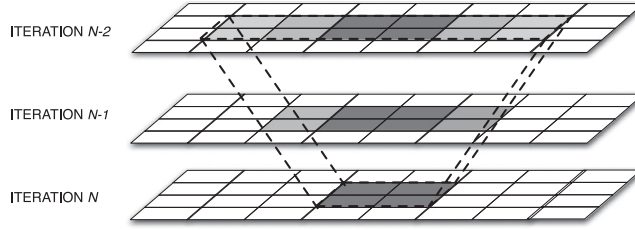


Fig. 3. 3D representation of a generic computational cone spanning two iterations.

The knowledge of the data dependencies makes it possible to express the result of the $(i + m)$ -th iteration as a function of (part of) the elements computed at the i -th iteration. As a consequence, given the data available from the i -th iteration, instead of trying to compute the whole f_{i+1} , the proposed approach focuses on a subset of the matrix elements and directly computes the results of a generic m -th iteration (with $m \geq 1$), thus obtaining a subset of f_{i+m} . The resulting computational cone has a *depth* equal to m .

To obtain the entire output frame f_{i+m} , multiple executions of the computational cones may be required. The proposed architectural template is defined as a combination of multiple levels of cones of different depths, which are able to compute the result of multiple iterations of the elementary transformation t . An instance of the proposed template is shown in Figure 4, and it works as follows: a small subset (*window*) of the input data—which is stored in the off-chip memory—is transferred to the on-chip memory to feed the cones of the first level of the architecture. In the example shown in Figure 4, the first level is composed of four cones: A, B, C, and D. The output of each level is then used as input for the subsequent level until all necessary iterations are performed. The output of the last level (Level 3 in the example in Figure 4) is finally stored back into the off-chip memory, and the whole process starts over on a different window of the input data, until all matrices have been computed. This technique, which allows spanning across the input matrix to progressively produce the output, is called *sliding window*.

The sliding window technique is illustrated more in detail in Figure 5. The windows are aligned in such a way that the correctly computed elements cover the entire frame, implying a certain degree of overlapping among them. The sliding windows approach introduces both a memory and a computation overhead. The former is due to the fact that certain elements are replicated in multiple submatrices and are processed by more than one cone. The latter is due to the structure of the cones, which are typically unaware of which part of the processed data is valid, and will eventually contribute to final output. The idea of dividing the input into a set of overlapping regions has already been proposed for a few specific algorithms in the scope of custom hardware design [Roca et al. 1999], even though it has never been methodically combined with other optimizations, such as the computation of multiple iterations within a cone.

Since the number and the depth of the cones in the actual architecture can vary depending on the desired trade-off among resources usage and target performance, multiple *instances* of the proposed template may exist. In particular, each one of these instances is uniquely defined by the two following parameters:

- (1) the size of the output window of each cone, defined as the number of output elements contained in the rectangle of size $l \times w$, and
- (2) the depth of each cone—for instance, the number of levels in which the computation is divided or, equivalently, the number of iterations that are performed at once by each cone.

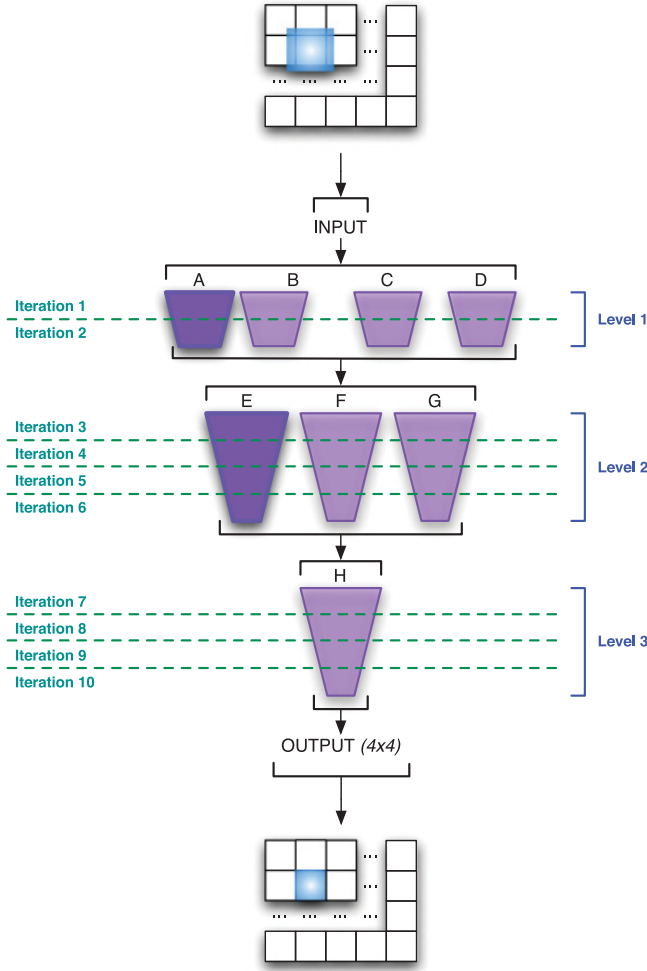


Fig. 4. An instance of the proposed cone-based architectural template.

Figure 4 shows an instance of the template with an output window of 4×4 elements and three levels of computation: the first one involves two iterations, whereas the other two levels involve four iterations each. It is worth noting that since the amount of data exchanged between two levels x and $x + 1$ (the output of level x is the input of level $x + 1$) only depends on the size of the output of level $x + 1$ and on the number of iterations considered by the two levels of computation, the parameters previously introduced suffice to completely specify any architecture.

The only requirement for an instance to be feasible is that if cones of different depths are required to complete the computation, at least one cone of each depth must be implemented on the device. For instance, the example in Figure 4 is feasible if the available resources are sufficient to fit cones A and E, because in this case, the first level can be implemented by sequentially executing cone A four times (to cover B, C, and D as well) and cone E four times (three executions are required for Level 2 and one for Level 3). Many instances are generally feasible, and the same instance may be implemented in different ways by instantiating different numbers of cores of different

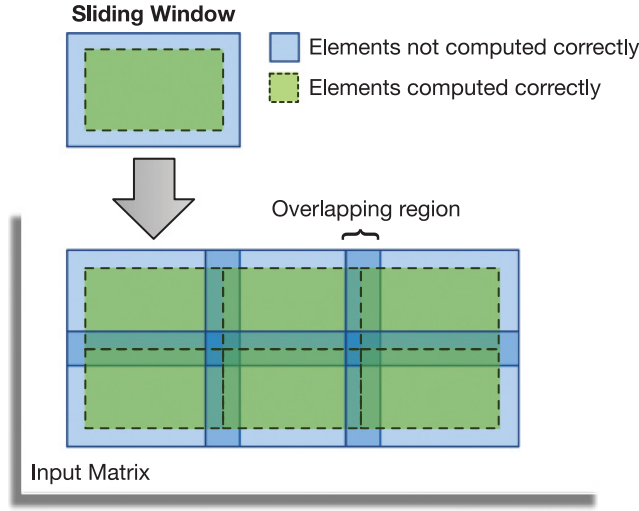


Fig. 5. The sliding window technique to produce the whole output frame.

depths, according to the availability of resources. As a consequence, multiple different trade-offs between area usage and achievable throughput (the more cones, the better) need to be evaluated. The trade-off analysis can be performed by defining proper quality metrics, which are discussed in the following section.

4.2. Design Evaluation Using the Expansion Rate

As the definition of a computational cone spanning across the frame introduces a computation and memory overhead in the final architecture, it is necessary to define proper quality metrics to estimate its impact and help the designer in tuning the architectural parameters, such as depth and window size of each cone. An ideal metric should only depend on the structure of the algorithm to be computed in the early stages of the design; however, on the other hand, it should provide a reliable estimation of postimplementation aspects, such as area and throughput. In this context, we define such a metric, related only to the geometry of the dependency scheme, and we name it *expansion rate*.

Two flavors of the expansion rate are proposed in this work: the first focuses on the geometry of the stencil, and the second is mainly driven by memory considerations. The two values are conceptually different, as they address two separate aspects of the design; hence, they can be considered as complementary while evaluating different design options. The two flavors of the expansion rate are defined as follows:

—*Static expansion rate (SER)*: The SER is defined as the normalized ratio between the number of input elements to be processed and the size of the output window. In particular, the SER for a cone of depth m that produces an output area of size $l \times w$ is defined as follows:

$$SER(l, w, m) = \sqrt[m]{\frac{\Omega(l, w, m)}{l \cdot w}}, \quad (16)$$

where $\Omega(l, w, m)$ is the set of input elements that must be processed to generate the output area while performing m iterations at once. The metric is purely based on geometrical considerations—in fact, $\Omega(l, w, m)$ only depends on the shape of the stencil, which in turn depends on the input algorithm. The m -th square root acts as

a normalization operation, which is necessary to compare cones of different depths. In fact, a cone with a higher depth likely requires a larger number of input elements to produce the same output area, but this higher overhead is compensated by the benefits of performing more iterations at once.

- Dynamic expansion rate (DER)*: The DER is conceptually defined as a ratio between the number of input elements that need to be loaded from the memory and the size of the output that is produced by the cone. The amount of data to be fetched from the memory is equal to the number of elements that are necessary to compute the current output window and were not required to compute the previous one. Hence, this metric is able to evaluate the overlapping of the sliding window and assess how this affects the memory access. Formally, the DER is defined as follows:

$$DER(l, w, m) = \frac{\Psi(l, w, m)}{l \cdot w}, \quad (17)$$

where the function $\Psi(l, w, m)$ indicates the number of nonoverlapping input elements between two consecutive applications of the cone. This value is specific for each input algorithm and can be computed by considering either a horizontal or a vertical translation of the sliding window.

The expansion rate is equal to 1 only if the output and the input window sizes are equal, and hence no overhead exists, whereas it assumes higher values when the number of input elements that are processed by the cone is much larger than the size of the output window. In this way, the expansion rate can be used to maximize the ratio between the number of output and of input elements. The metric is also a function of the depth of the cone, because performing a larger number of iterations at once reduces the number of intermediate results to be stored, increases performance, and may balance the additional overhead of processing a larger input window.

5. IMPLEMENTATION DETAILS

This section illustrates the design of a parallel implementation of Chambolle, whose structure is based on the cone architecture proposed in Section 4. Starting from the stencil shape of the algorithm, a set of cones have been derived and further optimized using ad hoc considerations. In particular, the design of the PEs within each cone has been specifically tuned to achieve the best possible performance, using an efficient and application-specific data reuse mechanism, described in Section 5.3, as well as a properly suited memory management system, detailed in Section 5.4. As a result of this design effort, the proposed solution largely outperforms all existing hardware implementations of Chambolle that can be found in the literature.

In the proposed architecture, the shape of the computational cone follows the stencil shape shown in Figure 2(b). Each cone aims at directly computing each element of px and py (see Algorithm 1) at iteration $n + x$ by finding a formula that employs the values available at iteration n . Each cone is then shifted using a sliding window mechanism to span the entire area of the input matrix. As discussed in Section 4, the rationale is to divide the output frame (I_1 in Section 3.4) into overlapping submatrices, whose profitable areas are contiguous. This approach introduces a slight memory overhead, because certain elements are replicated in multiple submatrices. A computation overhead is also introduced, as the cores may process some elements that are not profitable and will not be part of the output. However, the sliding window technique enables a coarse-grained parallelization of Chambolle despite its recursive nature and its complex data dependencies, and this greatly improves the throughput of the proposed implementation.

The remainder of this section provides a detailed description of the computation that takes place within each computational cone. In addition, we discuss the implementation

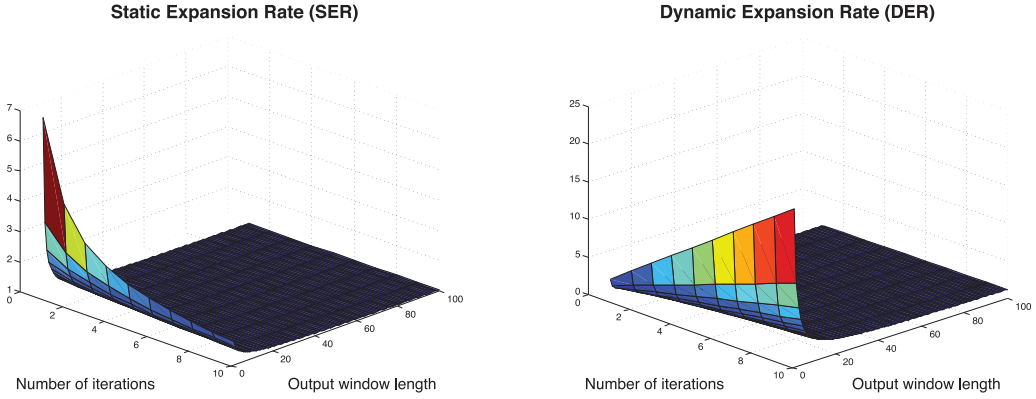


Fig. 6. Static and dynamic expansion rates for the Chambolle algorithm.

of the sliding window technique, which allows the cones to span the input matrix, including all relevant implementation details related to the memory organization.

5.1. Expansion Rate Analysis

The expansion rate metrics, which were introduced in Section 4, can be evaluated to guide the choice the most suitable cone size for the Chambolle algorithm.

The SER, which captures the geometrical properties of the algorithm, can be computed according to Equation (16), replacing the value of $\Omega(l, w, N)$ —which quantifies the number of input elements that must be processed to generate the output window—with the equation obtained in (15). The resulting equation is the following:

$$SER(l, w, m) = \sqrt{\frac{\sum_{k=1}^{m-1} \left[(l + 2k)(w + 2k) - 2 \sum_{h=1}^k h \right]}{l \cdot w}}. \quad (18)$$

This equation is plotted in Figure 6 for different values of the number of iterations and the output window size. For the sake of illustration, a squared output window has been assumed in the figure, so its size can be summarized using only one axis, which represents the length of its edge. It can be observed that the expansion rate is minimized with windows of large size (i.e., larger than 60×60), whereas a dependency with respect to the number of iterations is significant only for windows of small size. This behavior is consistent with the shape of the Chambolle stencil, which requires a lot of overlapping input elements when a large output is computed.

Similarly, the DER can be computed starting from Equation (17) and computing the number of elements to be fetched from the memory when the cone slides to the following output window. According to the shape of the stencil illustrated in Figure 2, the following can be derived:

- When the cone slides horizontally, a total of $l \cdot (w + 2m)$ new elements of the input matrix have to be fetched.
- When the cone slides vertically, $w \cdot (l + 2m)$ new elements have to be loaded from the memory.

The two sliding directions can be used indifferently to compute the DER, as they eventually lead to the same conclusions. Figure 6 shows the behavior of the DER for different values of the number of iterations and the output size: a squared output window is again assumed for illustrative purposes, thus making the horizontal and

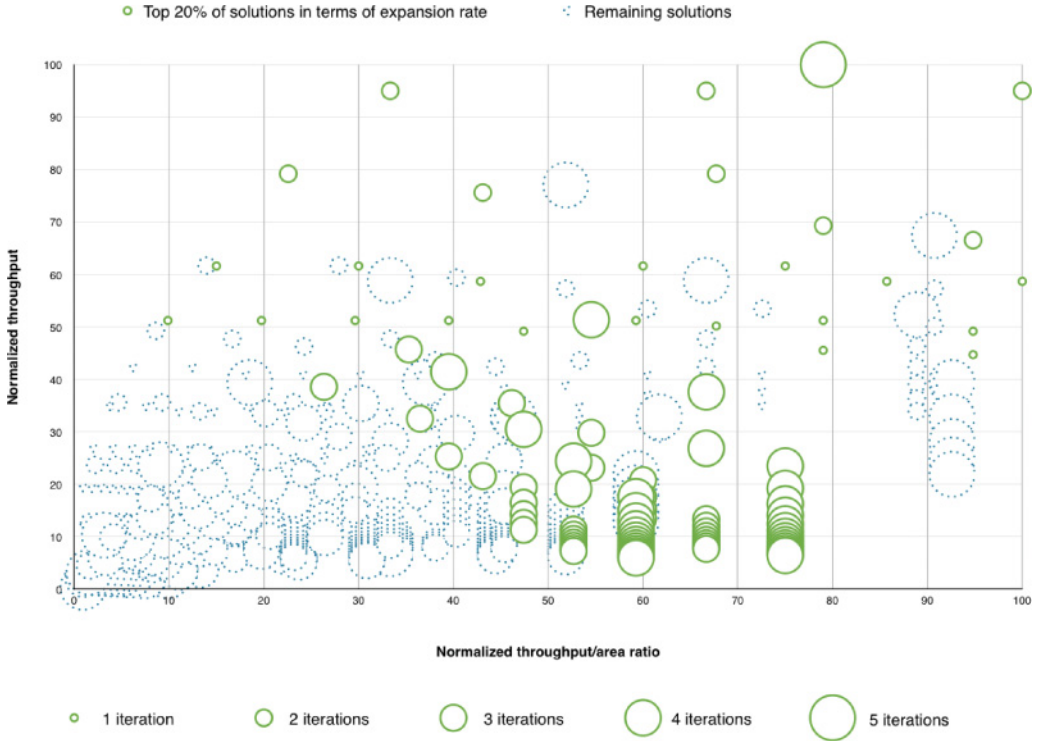


Fig. 7. Expansion rate estimation versus actual postimplementation area and throughput.

vertical translations equivalent. Similarly to the static case, the evaluation of the DER also recommends the employment of large output windows, with an edge larger than 80 elements.

The conclusion of the analysis of SER and DER, reported in Figure 6, is that a window whose length is larger than 60 and 80 elements should be preferred, respectively. The intersection of the two metrics ensures that any output window larger than 80×80 can effectively mitigate the effects of the computation and memory access overheads.

Finally, we use Chambole as an illustrative example to illustrate the ability of the expansion rate to capture postimplementation design aspects, specifically area and throughput, despite being defined as a sole function of the geometry of the input algorithm. Figure 7 highlights the best solutions when two common design approaches are adopted. Specifically, the x-axis represents the normalized ratio between throughput and area, which corresponds to a scenario where the design goal is to maximize the performance of the system, given the available resources. The y-axis, on the other hand, represents the normalized throughput, corresponding to a scenario where performance has to be maximized without area limitations. The quantitative analysis of Figure 7 includes different window sizes and number of iterations, which in turn correspond to different values of the expansion rate—in this case, the SER, but similar results are obtained for the DER. The window sizes range between 6×6 and 89×89 , whereas the number of iterations varies between 1 and 5, and is represented in the picture by the size of the circles. The green data points (solid lines) highlight the top 20% of solutions in terms of SER. It can be observed that in general, solutions with a higher expansion rate tend to have higher throughputs and make an efficient use of the area they require. This is further supported by the results in Figure 8, which reports throughput

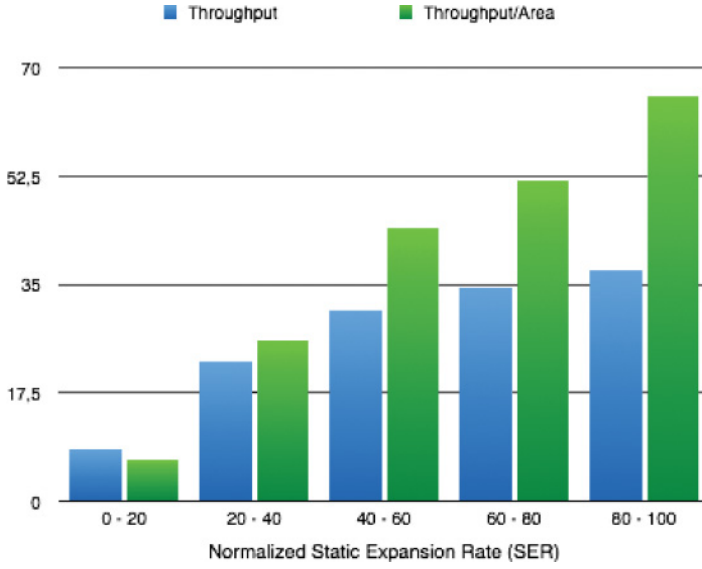


Fig. 8. Normalized throughput and throughput/area for different ranges of the normalized SER.

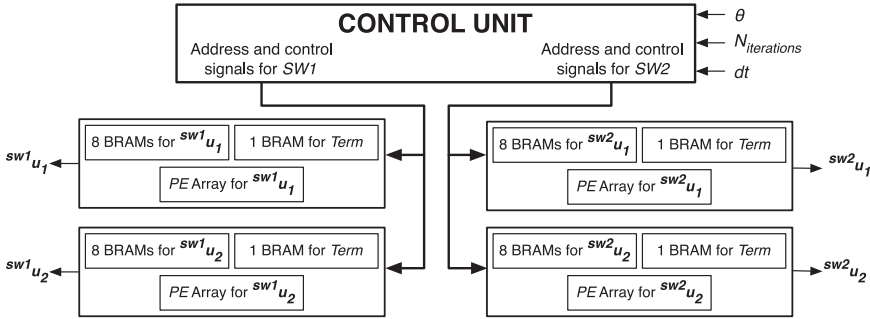


Fig. 9. Top-level block diagram of the proposed hardware implementation of Chambolle.

and throughput/area values as a function of the SER, the data points being clustered and averaged to better highlight the correlation. The expansion rate can therefore be considered as a reliable metric for design space exploration, and it can be computed by following the algorithm analysis proposed in Section 3 rather than performing a time-consuming synthesis for each candidate window size.

5.2. Overview of the Proposed Hardware Solution

Among the different implementations that satisfy the constraint identified in the previous section (windows larger than 80×80 elements), we herein propose as an example a solution that employs cones working on submatrices of 88×92 elements, which is close to the target threshold, to keep low resource (especially memory) requirements. The proposed hardware architecture slides these windows to span the entire length of the original matrix.

A top-level block diagram of the proposed hardware architecture is shown in Figure 9. The hardware employs two concurrent cones moving as sliding windows (named SW1 and SW2), which work completely in parallel, each one updating the values of both u_1

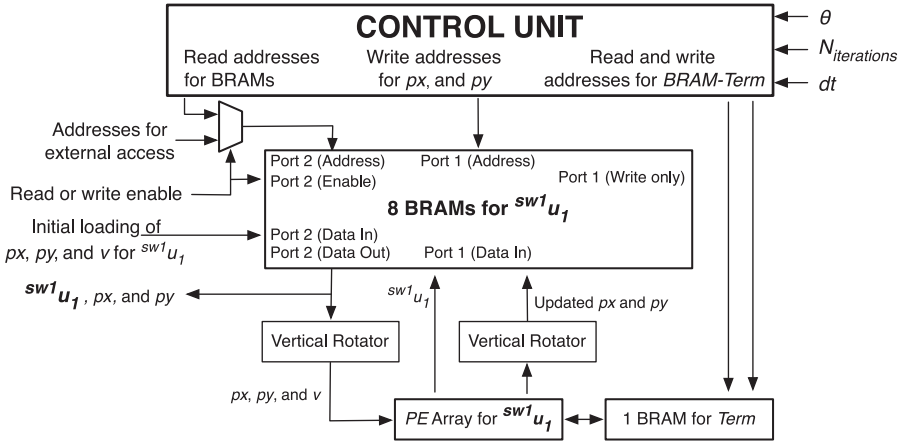


Fig. 10. Computation of $^{sw1}u_1$ within a cone.

and u_2 (we use the notation $^{sw1}u_1$ to indicate the value of u_1 computed by the sliding cone SW1). A cone moving as a sliding window is logically divided into two parts: an array of processing elements (PEs) and a dedicated amount of on-chip memory implemented on the BRAMs of the FPGA device.

A detailed view of a cone, and in particular of the circuit that processes $^{sw1}u_1$, is shown in Figure 10. The data required to compute the components of u (i.e., v , px , and py), as shown in Algorithm 1) is stored in the on-chip BRAMs to reduce the access to the off-chip memory. We have designed the cone to compute 7 elements in parallel for both u_1 and u_2 , thus finding 14 elements of vector u at the same time. This structure not only introduces a finer level of parallelism to accelerate the execution but also enables a significant data reuse among the PEs (as discussed in the following section) and reduces the access to both on-chip and off-chip memory.

As a result, the proposed hardware is able to compute the value of one element in just 18 clock cycles: 1 cycle is required by the control unit, 1 cycle by the synchronous read from the BRAM memory, 1 cycle by the vertical rotator, and 15 cycles by the PE array. Furthermore, the processing of each one of $^{sw1}u_1$, $^{sw1}u_2$, $^{sw2}u_1$, and $^{sw2}u_2$ requires 8 BRAMs to store the respective px , py , and v values, plus an additional BRAM that is necessary to exchange data between two iterations of the PEs. Hence, only 36 BRAMs blocks are employed by the proposed design.

5.3. Processing Element Arrays and Data Reuse

The proposed hardware implementation includes the proposed PE arrays, two for each cone, to find the outputs u_1 and u_2 of Chambolle, which are subsequently used to update v by means of the thresholding function. Each PE array contains 14 PEs, 7 of which are called *PE-Ts* and are used to calculate the values of *Term* and u (see Algorithm 1), whereas the other 7 are called *PE-Vs* and are used to compute px and py . Overall, there are 56 PEs in the proposed hardware, evenly divided among *PE-Ts* and *PE-Vs*.

Within the cone, a *ladder* organization of a PE array is proposed. Figure 11 illustrates this organization on the PEs that work on the first 7 rows (also called *first region*) of the input matrix. The same figure also illustrates how the same PEs are then reused to process the following seven rows (second region). In particular, while $PE-T_1$ is calculating *Term* for the elements in uppermost row, $PE-T_7$ computes *Term* for the elements in row 6. Then, after all PEs have completed the first seven rows, $PE-T_1$ starts computing *Term* for row 7 while $PE-T_7$ shifts to row 13.

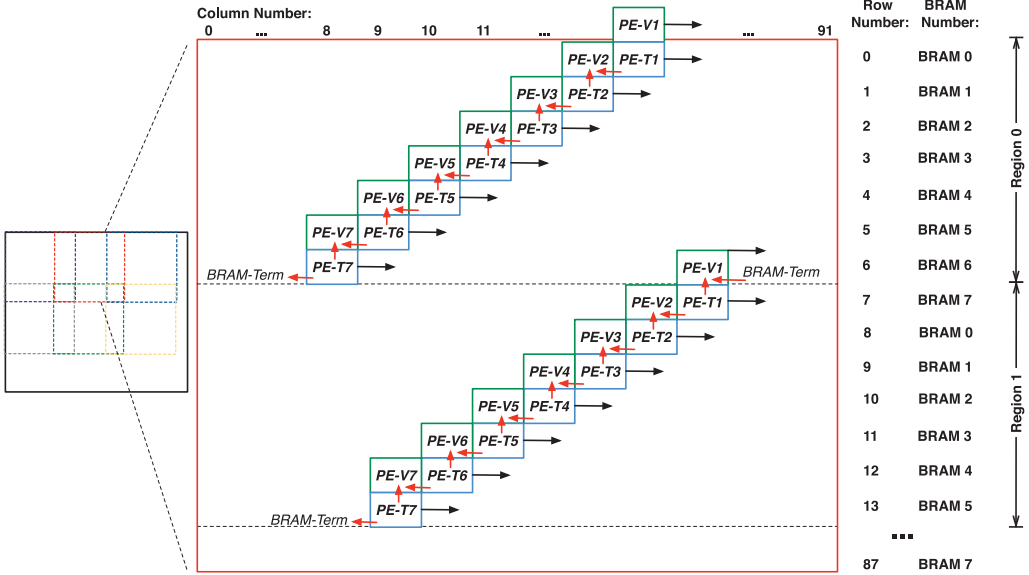


Fig. 11. Organization of seven PE -Ts and seven PE -Vs in a computational cone, and memory organization during the computation of $^{sw1}u_1$.

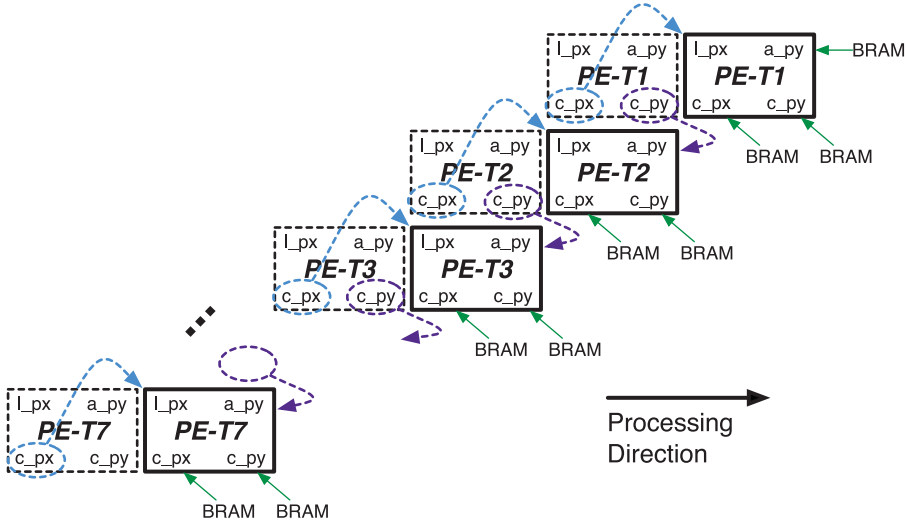


Fig. 12. Data reuse among the seven PE -Ts during the computation of $^{sw1}u_1$ (the dashed boxes indicate the position of the PE -Ts in the previous cycle).

The value of $Term$ for one element depends on the values of px and py at the same position (we refer to these values as c_px and c_py), plus the px vector of the element on the left (l_px) and the py vector of the element above (a_py). Without any data reuse policy, each PE -T in a PE array requires 4 values to be loaded from the on-chip memory, and consequently four PE arrays with 7 PE -Ts require 112 values to be read from the memory. Thanks to the proposed ladder organization of the PE s, this data transfer can be limited by propagating the intermediate results. Figure 12 shows how the seven PE -Ts are disposed and how they were aligned in the previous cycle (dashed boxes).

Since all PEs require their c_px and c_py vectors to be computed in a previous iteration, they are loaded from the BRAMs. Then, as the processing direction in a cone goes from left to right, these vectors can be reused as l_px and a_py vectors for the following cycle without accessing the memory. For instance, $PE-T_3$ takes the l_px vector from the flip-flop that stores the c_px vector processed in previous cycle. Similarly, c_py can be reused as a_py by the $PE-T$ s that are located below, as for example the c_py vector used by $PE-T_2$ is the a_py vector of $PE-T_3$ for the next cycle.

The $PE-V$ s start computing px and py for one element one cycle after the $PE-T$ s, and they also exploit a massive reuse of data. Algorithm 1 shows that to compute px and py vectors for an element, three $Term$ values are required: the one of the corresponding element, the one of its right neighbor, and the one of the bottom neighbor. In the proposed implementation, the values of $Term$ that are processed by the array of $PE-T$ s are reused and propagated using pipelining flip-flops. For instance, to compute px and py for the element at position (2, 11), the $Term$ values of elements in (2, 11), (2, 12), and (3, 11) are required. $PE-T_3$ calculates the $Term$ value at (2, 11), and at the same time $PE-T_4$ calculates the $Term$ value for (3, 10). In the next clock cycle, $PE-T_3$ and $PE-T_4$ compute the $Term$ values for (2, 12) and (3, 11), respectively. Then, $PE-V_3$ takes the required $Term$ values from $PE-T_3$ and $PE-T_4$, as well as the synchronized result of $PE-T_3$ that was computed in previous clock cycle, and determines the new px and py for element (2, 11) without reading any data from BRAM. Once the values of px and py have been determined, they are stored in BRAM for the following iterations.

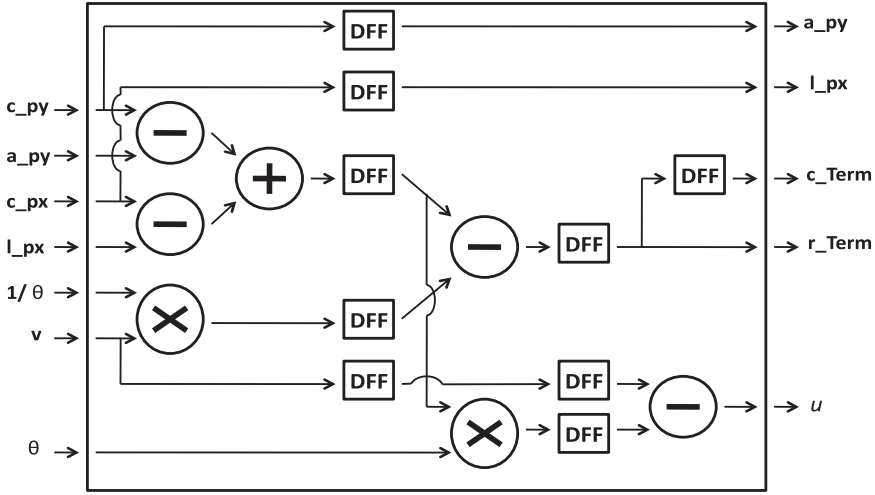
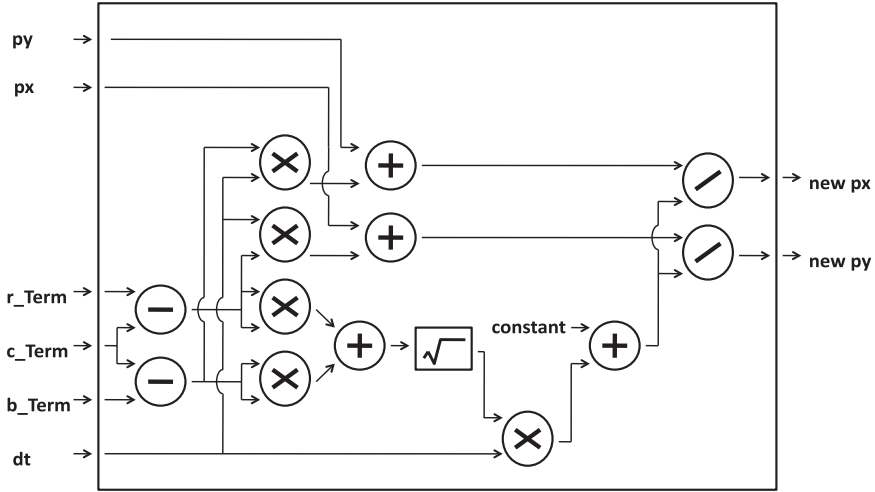
5.4. Memory Organization

The proposed data reuse scheme reduces both the number of accesses to the BRAMs and the amount of memory required to store the intermediate results. As shown in Figure 12, the array of $PE-T$ s needs to read 15 vectors from BRAMs, but 28 vectors would be required if data reuse had not been implemented. We now illustrate how those BRAMs are organized.

According to Figure 11, $PE-V$ s from 2 to 7 take the required values of $Term$ from the two adjacent $PE-T$ s and from the result computed in the previous clock cycle by the $PE-T$ s that are on their right. Therefore, the computation of these six $PE-V$ s does not require any additional BRAM to store the intermediate values of $Term$ computed by the $PE-T$ s. Only $PE-V_1$ needs to load the $Term$ values computed by $PE-T_7$ in the previous region, which has to be stored in a BRAM block (called $BRAM-Term$). For instance, to calculate px and py for row 6, the values of $Term$ for rows 6 and 7 are required, but they cannot be computed in successive clock cycles because the two rows belong to two different regions (see Figure 11) and are processed by the PE array in two separate moments. Therefore, the $Term$ values of row 6 are stored in a dual-port BRAM, and they are read back when $PE-T_1$ computes the $Term$ values of row 7.

As a PE uses eight BRAMs for px , py , and v , plus an additional $BRAM-Term$ block as a bridge between two different regions, nine BRAMs are required to process each region. The results computed by each $PE-V$ are stored in the corresponding BRAMs according to the addressing shown in Figure 11. When the array completes a region and starts processing the following one, the address used to access the BRAMs needs to be increased by an offset of 92, and this step is performed by a *vertical rotator*, which is shown in Figure 10.

Overall, the eight BRAMs of each region are indexed using 1,012 addresses, and 32 bit blocks of data are stored in each address. The 32 bits encode v , which requires 13 bits, followed by c_px and c_py , which require 9 bits each. After the $PE-V$ s find the new values of px and py , the values in the BRAMs are updated by using the write ports of the BRAMs, overwriting the vector values that have been read in previous cycles.

Fig. 13. Hardware architecture of a *PE-T*.Fig. 14. Hardware architecture of a *PE-V*.

5.5. Processing Elements

We finally provide a detailed description of the *PE-T* and *PE-V* PEs. The hardware architecture of a *PE-T* is shown in Figure 13, and the one of a *PE-V* is shown in Figure 14.

The implementation of a *PE-T* includes the *Backward* operations for px and py , which are performed in parallel before computing the value of the output *Term*, which is then used as r_Term (*right Term*) for the *PE-V* that is processing the same row, whereas b_Term (*bottom Term*) can feed the *PE-V* that is processing the upper row. Moreover, the value of *Term* is pipelined for one clock cycle to use it as c_Term (*current Term*). The propagation schema of the different *Terms* (*right*, *bottom*, and *current*) is shown in Figure 11.

The hardware architecture for *PE-Vs* implements the *Forward* operations between c_Term , r_Term , and b_Term in parallel and then computes the new px and py vectors.

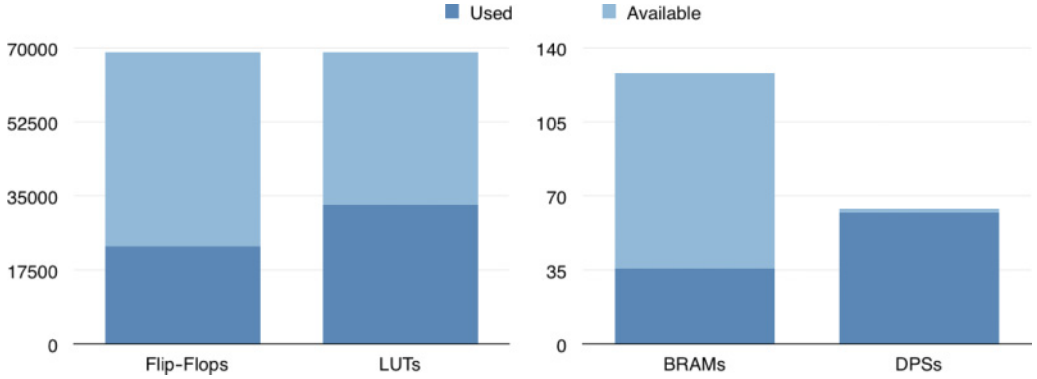


Fig. 15. Area usage on a Xilinx Virtex-5 XC5VLX110T FPGA.

The main issue in the design of the *PE-V* architecture is the square root function to compute px and py , as shown on line 6 of Algorithm 1. An efficient and precise hardware implementation of the square root is still an open problem [Sajid et al. 2010; Li and Chu 1997], and there are two main techniques to handle it: iterative techniques, which achieve better precisions, and look-up tables (LUTs), which are faster.

In the proposed implementation, an LUT implementation was employed to focus on timing performance, whereas the achieved precision is still acceptable in the context of optical flow estimation. In fact, the error of the approximated square root is below 1% in more than 90% of the tested samples. The LUT takes a 32-bit signal represented using a fixed point notation, where the integer part takes 24 bits and the decimal part takes 8 bits. The entries of the table are 8-bit values, and thus the table contains $2^8 = 256$ precomputed values and only requires 70 LUTs to be deployed on the FPGA. Instead of dividing the input value into four pieces of 8 bits each, which can index four different tables, a technique has been designed to increase the precision while using only one table (thus saving approximately 12,200 LUTs over the 28 *PE-V*s). In particular, the 8 most significant bits of the input value are considered and used to get the result from the table, discarding the remaining bits. The 8-bit block starts in an odd position (counting from left to right) and finishes in an even one: if the first nonzero bit is located in the n -th position, where n is even, then the 8-bit block will start from the zero bit at position $n - 1$. In this way, if the decimal value of the 8-bit block is equal to m , and if the rightmost bit of the block is in position $2k$, then the number is equal to $m \cdot 2^{2k}$, and its square root is computed by accessing the table at value m and left-shifting the output by k positions.

6. EXPERIMENTAL RESULTS

The proposed cone-based parallelization of the Chambolle algorithm has been fully implemented in Verilog and synthesized for a Xilinx Virtex-5 XC5VLX110T FPGA [Xilinx 2009]. Figure 15 shows the resource usage of the Chambolle core, which reaches an operating frequency of 221MHz after place and route. If required by the target device, the number of required DSPs can be reduced by mapping part of the multiplications on the LUTs.

Figure 16 shows the comparison, in terms of frames per second, between the performance achieved by the proposed approach and the ones obtained by state-of-the-art implementations. These are implemented on either CPUs or GPUs as, to the best of our knowledge, no implementation that leverages the fine-grained parallelism of FPGAs has been proposed in the literature. The evaluation assumes that the images to

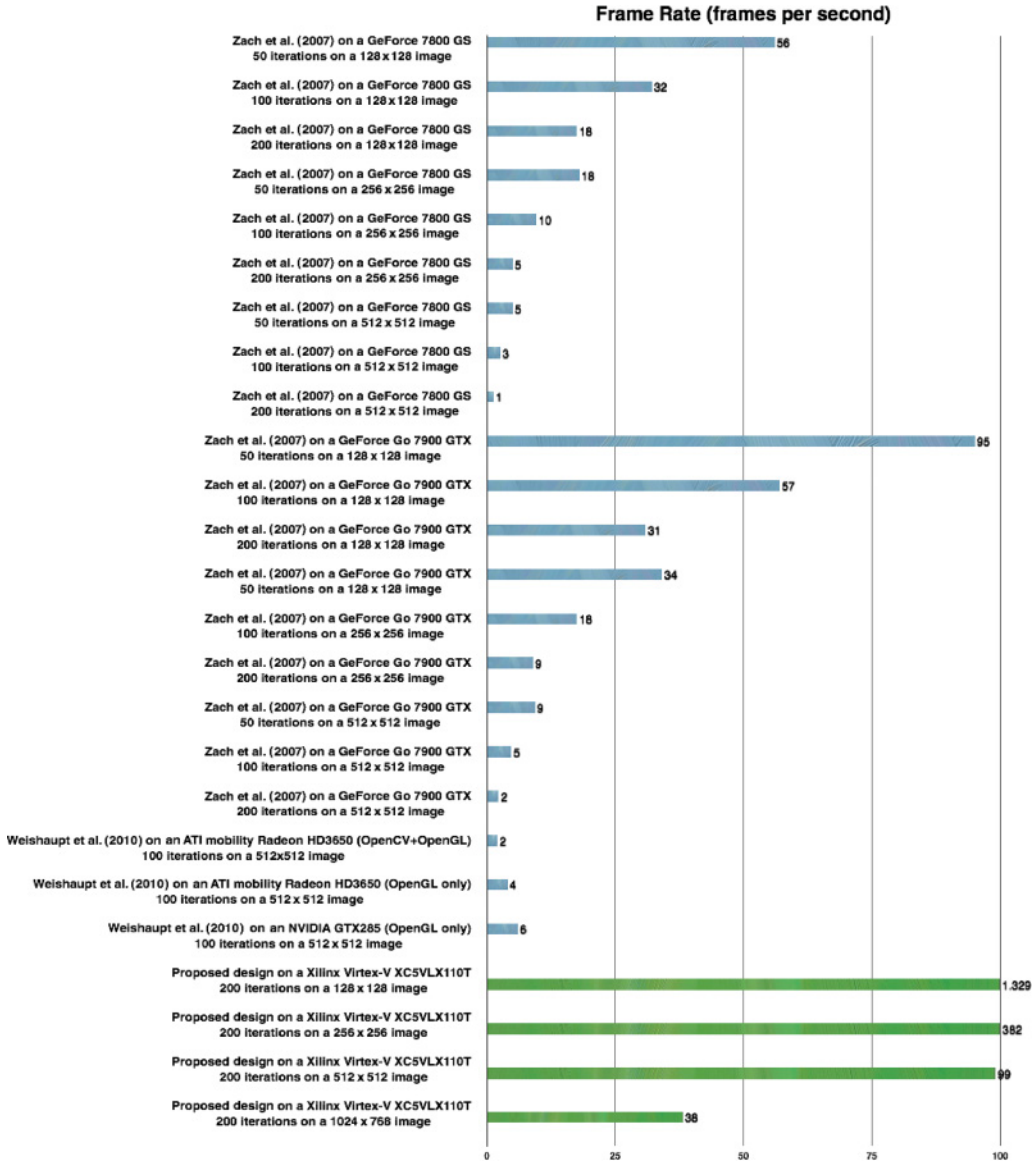


Fig. 16. Performance comparison, in terms of frames per second, with respect to state-of-the-art implementations.

be processed are preloaded in the device memory to focus the measurements on the Chambolle algorithm itself rather than on the transient setup. The estimated speedup achieved by the implementation proposed in this work ranges from $16.5\times$ to $76\times$ on images with a resolution of 512×512 , which is the most common format found in the literature related to Chambolle.

However, the advantages of the proposed parallelization approach are even more noticeable on larger images. In fact, the proposed implementation is the only one able to achieve more than 30fps—and hence meet the real-time constraints—on 1024×768 images. On the contrary, most of the existing approaches work with reasonable frame

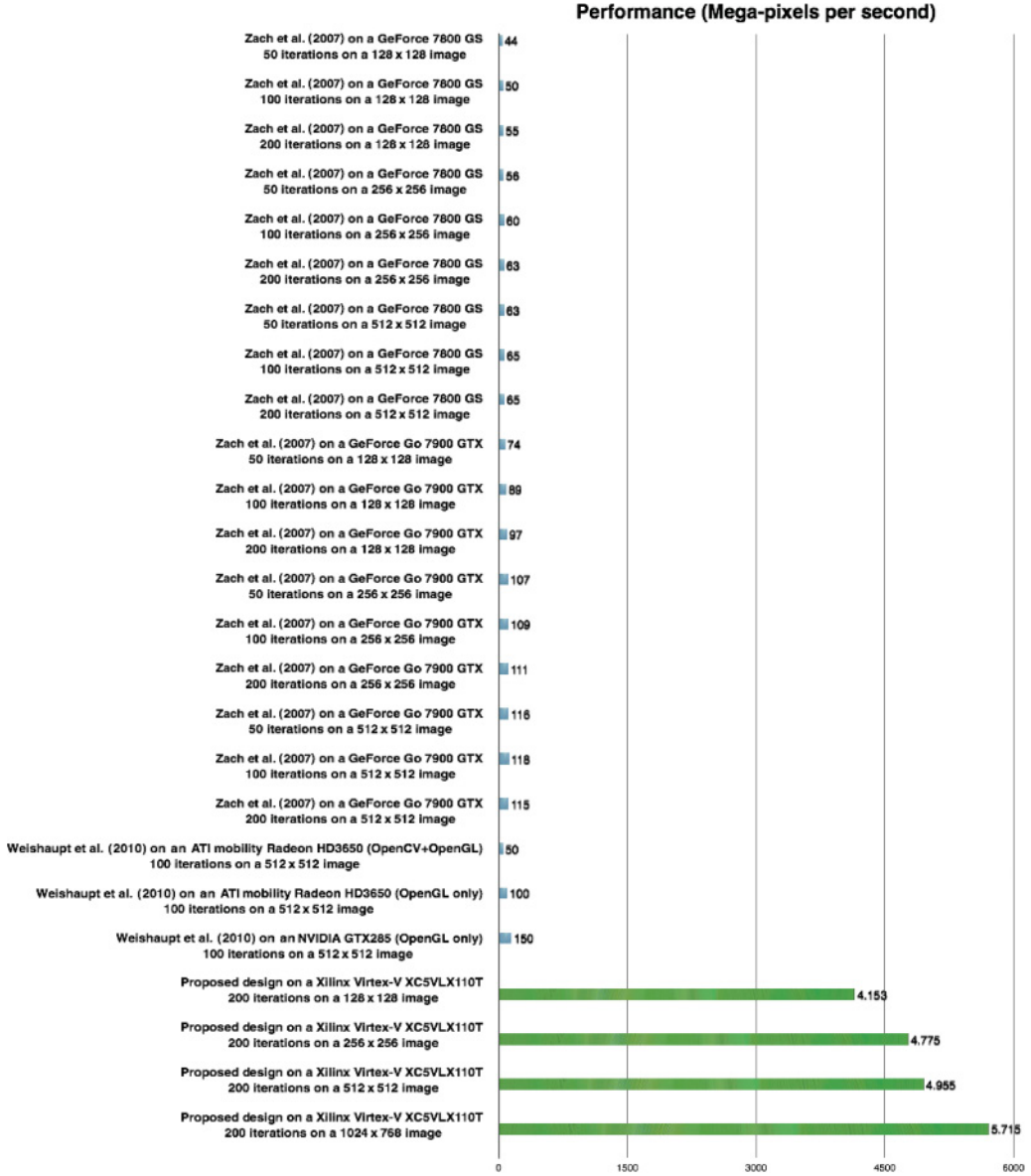


Fig. 17. Performance comparison, in terms of mega-pixels per second, with respect to state-of-the-art implementations.

rates (higher than 20fps) only on very small images (consisting of either 128×128 or 256×256 pixels). Thus, to perform a fair comparison and to normalize the size of the images processed by the different approaches, we compare them in Figure 17 in terms of number of megapixels elaborated per second. In this case, the speedup obtained by the proposed design with respect to the best state-of-the-art implementations ranges from $38\times$ to $130\times$ ($77\times$ in the average), proving that the proposed approach scales very well with the frame size.

7. COMPARISON WITH RESPECT TO GPU IMPLEMENTATIONS

We finally discuss a possible implementation of Chambolle on GPUs to prove how the fine-grained configuration capabilities of FPGAs provide a better environment for the implementation of this algorithm. Comparisons among the two architectures have been already proposed in the literature, such as in Bodily et al. [2010], proving that GPUs do not match the flexibility provided by FPGAs when custom computation is required. A similar discussion is herein performed in the context of Chambolle, since the algorithm has been analyzed in depth in the previous paragraphs, and all of its details—from the mathematical formulation to the actual FPGA implementation—are known at this point.

The GPU framework considered as a potential target for Chambolle is *CUDA* [NVIDIA 2007] by NVIDIA. Following the architectural model of these GPUs, applications are divided into parallel portions that are executed on the device as *kernels*, which are in turn implemented by a grid of independent *blocks* that execute a set of *threads*. The memory hierarchy consists of three levels: a *local memory* is used by each thread, an on-chip *shared memory* is used within a block to exchange data and synchronization information among the threads, and finally a *global memory* is used among consecutive kernels. The modern NVIDIA Fermi architecture [NVIDIA 2009] features 512 cores divided into 16 *streaming multiprocessors* (SMs) of 32 cores each. The interesting feature of an SM is the availability of a unit to load and store data from the two-level cache memory and DRAM, and of a set of special function units, including one for the inverse and one for the square root of a number.

To implement Chambolle on the Fermi GPU architecture, a mapping of the operations on the thread blocks is required, as well as ad hoc memory considerations. In the proposed parallelization of the algorithm, each element of the matrix requires only the elements that it caches to complete its computation, and therefore the elaboration of a single element can be assigned to a separate thread. As a consequence, the elaboration of a window can be assigned to a single CUDA block and, using the sliding window technique, more blocks cover the entire frame. However, because of the fixed structure of the architecture, only 64 elements of the input matrix can be processed in a single SM. Given the availability of 16 SMs, 16 windows can be processed in parallel, thus allowing the concurrent computation of $16 \times 64 = 1024$ elements. The latter value is considerably lower than the FPGA counterpart—in which each cone could process $88 \times 92 = 8,096$ elements at once—and it translates a higher overhead in terms of data that needs to be transferred from the memory. This inefficient parallelization is only partially compensated by the higher frequencies of GPUs, as new data cannot be produced at each clock cycle because of the presence of difficult operations like the square root, which itself requires eight clock cycles. On FPGAs, on the other hand, the possibility of customizing the structure of the computational cone leads to a more efficient and tailored design, in which operations such as the square root can be arbitrarily optimized and approximated according to the application requirements.

8. CONCLUDING REMARKS

After introducing the Chambolle algorithm and describing its main features, we performed a deep analysis on its structure and provided a formal proof of the locality of its dependency schema. We then exploited the considerations derived by this analysis to propose a novel template architecture that exploits the implicit fine-grained parallelism that can be extracted by this kind of multimedia algorithms. However, since the proposed template can be instantiated with different parameters, we have also introduced a metric, called *expansion rate*, to help the designer in the exploration of the solution space. The proposed analysis and parallelization approach, applied to the

Chambolle algorithm, have been proven to be effective and able to generate efficient FPGA-based computing architectures, of which performance is orders of magnitude faster than the state-of-the-art ones, when compared on the number of megapixels produced per second.

REFERENCES

- M. M. Abutaleb, A. Hamdy, M. E. Abuelwafa, and E. M. Saad. 2009. A reliable FPGA-based real-time optical-flow estimation. In *Proceedings of the National Radio Science Conference (NRSC'09)*. 1–8.
- Abdulkadir Akin, Ivan Beretta, Alessandro Antonio Nacci, Vincenzo Rana, Marco Domenico Santambrogio, and David Atienza. 2011. A high-performance parallel implementation of the Chambolle algorithm. In *Proceedings of the IEEE/ACM 2011 Design, Automation, and Test in Europe Conference (DATE'11)*. 7–12.
- Karim M. A. Ali, Rabie Ben Atitallah, Said Hanafi, and Jean-Luc Dekeyser. 2014. A generic pixel distribution architecture for parallel video processing. In *Proceedings of the 2014 International Conference on ReConfigurable Computing and FPGAs (ReConFig'14)*. 1–8. DOI: <http://dx.doi.org/10.1109/ReConFig.2014.7032547>
- G. Aubert, R. Deriche, and P. Kornprobst. 1999. Computing optical flow via variational techniques. *SIAM Journal on Applied Mathematics* 60, 156–182.
- Simon Baker, Eric P. Bennett, Sing Bing Kang, and Richard Szeliski. 2010. Removing rolling shutter wobble. In *Proceedings of the 2010 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'10)*. 2392–2399.
- S. Behbahani, S. Asadi, M. Ashtiyani, and K. Maghooli. 2007. Analysing optical flow based methods. In *Proceedings of the 2007 IEEE International Symposium on Signal Processing and Information Technology*. 133–137. DOI: <http://dx.doi.org/10.1109/ISSPIT.2007.4458079>
- M. J. Black and P. Anandan. 1993. A framework for the robust estimation of optical flow. In *Proceedings of the 4th International Conference on Computer Vision*. 231–236. DOI: <http://dx.doi.org/10.1109/ICCV.1993.378214>
- John Bodily, Brent Nelson, Zhaoyi Wei, Dah-Jye Lee, and Jeff Chase. 2010. A comparison study on implementing optical flow and digital communications on FPGAs and GPUs. *ACM Transactions on Reconfigurable Technology and Systems* 3, 2, Article No. 6. DOI: <http://dx.doi.org/10.1145/1754386.1754387>
- Antonin Chambolle. 2004. An algorithm for total variation minimization and applications. *Journal of Mathematical Imaging and Vision* 20, 1–2, 89–97. DOI: <http://dx.doi.org/10.1023/B:JMIV.0000011325.36760.1e>
- Peng Chen, Donglei Yang, Weihua Zhang, Yi Li, Binyu Zang, and Haibo Chen. 2012. Adaptive pipeline parallelism for image feature extraction algorithms. In *Proceedings of the 2012 41st International Conference on Parallel Processing (ICPP'12)*. 299–308. DOI: <http://dx.doi.org/10.1109/ICPP.2012.14>
- D. Cordes, M. Engel, O. Neugebauer, and P. Marwedel. 2013. Automatic extraction of pipeline parallelism for embedded heterogeneous multi-core platforms. In *Proceedings of the 2013 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'13)*. 1–10. DOI: <http://dx.doi.org/10.1109/CASES.2013.6662508>
- R. Ghodhban, T. Saidani, L. Horrigue, and M. Atri. 2014. Analysis and implementation of parallel causal bit plane coding in JPEG2000 standard. In *Proceedings of the 2014 World Congress on Computer Applications and Systems (WCCAS'14)*. 1–6. DOI: <http://dx.doi.org/10.1109/WCCAS.2014.6916602>
- Berthold K. P. Horn and Brian G. Schunk. 1981. Determining optical flow. *Artificial Intelligence* 17, 185–203.
- E. Jamro and K. Wiatr. 2001. Convolution operation implemented in FPGA structures for real-time image processing. In *Proceedings of the 2nd International Symposium on Image and Signal Processing and Analysis (ISPA'01)*. 417–422. DOI: <http://dx.doi.org/10.1109/ISPA.2001.938666>
- Guo-An Jian, Jui-Sheng Lee, Kheng-Joo Tan, Peng-Sheng Chen, and Jiun-In Guo. 2013. A real-time parallel scalable video encoder for multimedia streaming systems. In *Proceedings of the 2013 International Symposium on VLSI Design, Automation, and Test (VLSI-DAT)*. 1–4. DOI: <http://dx.doi.org/10.1109/VLSI-DAT.2013.6533845>
- Sungbok Kim, Ilhwa Jeong, and Sanghyup Lee. 2007. Mobile robot velocity estimation using an array of optical flow sensors. In *Proceedings of the International Conference on Control, Automation, and Systems (ICCAS'07)*. 616–621. DOI: <http://dx.doi.org/10.1109/ICCAS.2007.4407097>
- Yamin Li and Wanming Chu. 1997. Implementation of single precision floating point square root on FPGAs. In *Proceedings of the 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. 226–232. DOI: <http://dx.doi.org/10.1109/FPGA.1997.624623>

- S. Lin, Y. Q. Shi, and Y.-Q. Zhang. 1997. An optical flow based motion compensation algorithm for very low bit-rate video coding. In *Proceedings of the 1997 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP'97)*, Vol. 4. 2869–2872. DOI: <http://dx.doi.org/10.1109/ICASSP.1997.595388>
- NVIDIA. 2007. NVIDIA CUDA Compute Unified Device Architecture Programming Guide. Available at <http://www.nvidia.com>.
- NVIDIA. 2009. NVIDIA Next Generation CUDA Compute Architecture: Fermi. Available at <http://www.nvidia.com>.
- Nils Papenberg, Andrés Bruhn, Thomas Brox, Stephan Didas, and Joachim Weickert. 2006. Highly accurate optic flow computation with theoretically justified warping. *International Journal of Computer Vision* 67, 2, 141–158. DOI: <http://dx.doi.org/10.1007/s11263-005-3960-y>
- T. Pock, M. Urschler, C. Zach, R. Beichel, and H. Bischof. 2007. A duality based algorithm for TV- L^1 -optical-flow image registration. *Med Image Computing and Computer Assisted Intervention* 10, 511–518. <http://www.ncbi.nlm.nih.gov/pubmed/18044607>.
- Elisenda Roca, Servando Espejo, Rafael Dominguez-Castro, Gustavo Linan, and Angel Rodriguez-Vazquez. 1999. A programmable imager for very high speed cellular signal processing. *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology* 23, 2–3, 305–318. DOI: <http://dx.doi.org/10.1023/A:1008193018623>
- Leonid I. Rudin, Stanley Osher, and Emad Fatemi. 1992. Nonlinear total variation based noise removal algorithms. *Physica D: Nonlinear Phenomena* 60, 1, 259–268. DOI: [http://dx.doi.org/10.1016/0167-2789\(92\)90242-F](http://dx.doi.org/10.1016/0167-2789(92)90242-F)
- I. Sajid, M. M. Ahmed, and S. G. Ziavras. 2010. Pipelined implementation of fixed point square root in FPGA using modified non-restoring algorithm. In *Proceedings of the 2010 2nd International Conference on Computer and Automation Engineering (ICCAE'10)*, Vol. 3. 226–230. DOI: <http://dx.doi.org/10.1109/ICCAE.2010.5452039>
- Gerard L. G. Sleijpen and Henk A. Van Der Vorst. 2000. A Jacobi-Davidson iteration method for linear eigenvalue problems. *SIAM Journal on Matrix Analysis and Applications* 17, 401–425.
- S. Sun, D. Haynor, and Yongmin Kim. 2000. Motion estimation based on optical flow with adaptive gradients. In *Proceedings of the 2000 International Conference on Image Processing*, Vol. 1. 852–855. DOI: <http://dx.doi.org/10.1109/ICIP.2000.901093>
- A. Verri and T. Poggio. 1989. Motion field and optical flow: Qualitative properties. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 11, 5, 490–498. DOI: <http://dx.doi.org/10.1109/34.24781>
- Andreas Weishaupt, Luigi Bagnato, Emmanuel D'Angelo, and Pierre Vanderghenst. 2010. *Tracking and Structure from Motion*. Technical Report. École Polytechnique Fédérale de Lausanne (EPFL). <http://infoscience.epfl.ch/record/146572>.
- Xilinx. 2009. Virtex-5 Family Overview, DS100 (v5.0). Available at <http://www.xilinx.com>.
- C. Zach, T. Pock, and H. Bischof. 2007. A duality based approach for realtime TV- L^1 optical flow. In *Proceedings of the 29th DAGM Conference on Pattern Recognition*. 214–223. <http://dl.acm.org/citation.cfm?id=1771530.1771554>

Received February 2015; revised October 2015; accepted November 2015